Analysis of Software Complexity Measures for Regression Testing

Mrinal Kanti Debbarma[1], Nagendra Pratap Singh[2], Amit Kr. Shrivastava[3], and Rishi Mishra[4]
Computer Science & Engineering Department
MNNIT Allahabad, INDIA
mkdbarma@yahoo.com,{nagendrasngh447,amu02303,rishi.msr}@gmail.com

***Abstract*— Software metrics is applied evaluating and assuring software code quality, it requires a model to convert internal quality attributes to code reliability. High degree of complexity in a component (function, subroutine, object, class etc.) is bad in comparison to a low degree of complexity in a component. Various internal codes attribute which can be used to indirectly assess code quality. In this paper, we analyze the software complexity measures for regression testing which enables the tester/developer to reduce software development cost and improve testing efficacy and software code quality. This analysis is based on a static analysis and different approaches presented in the software engineering literature.**

***Index Terms*—Software Complexity, Software Metrics, Regression Testing, Control Flow Metrics[1].**

I. Introduction

The Software complexity is based on well-known software metrics, this would be likely to reduce the time spent and cost estimation in the testing phase of the software development life cycle (SDLC), which can only be used after program coding is done. The path count complexity of a function is defined as product of the path complexity of individual constructions. Improving quality of software is a quantitative measure of the quality of source code. This can be achieved through definition of metrics, values for which can be calculated by analyzing source code or program is coded. A number of Software measures widely used in the software industry are still not well understood [2]. Although some software complexity measures were proposed over thirty years ago and some others proposed later. Sometimes software growth is usually considered in terms of complexity of source code. Various metrics are used, which unable to compare approaches and results. Not all metrics are similarly easy to calculate for a given source code.[4]. Software systems are maintained by designers by doing regression test periodically in expect to find bugs caused by modifications and hoping that modifications made in the software are correct. Software engineering goal is to measure different aspects of software projects, to find small set of attributes that may characterize them. This paper presents an approach by which tester/developer can reduce software development cost and improve testing efficacy and software quality.

II. Background Details

A. *Regression Testing*

Regression testing is a process that seeks to uncover errors, which is performs after changes are made to program and can be used before release of modified program. Regression test can be perform rerunning the existing test suites against modified program whether the changes are correct and have no effect to unchanged part of the program. Regression test can be performing with adequate coverage that should be the primary consideration. A regression test compares the operation of the new version of software to the operation of a older version. The key idea is that the behavior of the program should not change in unanticipated ways. Regression testing ensures that we do not introduce new bugs or resurrect old ones. Testing of software is an integral part and key component of the software development process.This testing process can be used to test a system efficiently by selecting minimum set of test suite needed to that change. Regression testing techniques can be described as follows: Let P be a program, and P' be a modified program of P, and T be a test suit for program P. Regression testing can be attempt to revalidate modified program P'. From software engineering point of view software development experience shows, that it is difficult to set measurable targets when developing software products. Produced/developed software has to be testable, reliable and maintainable. On the other side, "You cannot control what you cannot measure"[3]. To avoid this, regression testing is performed during changes are made to existing software; the purpose of regression testing is to provide modified program do not obstruct existing, unchanged part of the software [1].Complexity of software is measuring of code quality; it requires a model to convert internal quality attributes to code reliability. High degree of complexity in a component (function, subroutine, object, class etc.) is bad in comparison to a low degree of complexity in a component is considered good. Software complexity measures which enables the tester to counts the acyclic execution paths through a component and improve software code quality. In a program characteristic that is one of the responsible factors that affect the developer's productivity [6] in program comprehension, maintenance, and testing phase. There are several methods to calculate complexity measures were investigated, e.g. different version of LOC [6], NPATH [7], McCabe's cyclomatic number [10], Data quality [10], Halstead's software science [8] etc.

[1] This research work was carried out at the Department of Computer Science & Engineering, MNNIT, Allahabad (UP), India-211004.
Corresponding author : mkdbarma@yahoo.com

DOI: 01.IJIT.01.02.158

B. Software Complexity Metrics: Properties

Complexity of software can be measure by selected properties that cause complexity. Some properties influencing the complexity are as follows:

- Size metrics
- Control flow metrics

1. Size metrics :Lines of Code

The size of the program indicates the development complexity, which is known as Lines of Code (LOC). The simplest measure of complexity recommended by Hatton (1977) . This metric is very simple to use and measure the number of source instruction required to solve a problem. While counting a number of instructions (source), line used for blank and commenting lines are ignored. The size, complexity of today's software systems demands the application of effective testing techniques. Size attributes are used to describe physical magnitude, bulk etc. Lines of code and Halstead's software science[8] are examples of size metrics. M. Halstead proposed a metrics called software science [Halstead 77].

2. Control Flow metrics: NPATH

The control flow complexity metrics are derived from the control structure of a program. The control flow measure by NPATH, invented by Nejmeh [7] ,it measures the acyclic execution paths, NPATH is a metric which counts the number of execution path through a functions. NPATH is example of control flow metrics. One of the popular software complexity measures NPATH complexity (NC) is determined as:

$NPATH = \prod_{i=1}^{i=N} NP(\text{statement}_i)$

NP (if) = NP(expr) + NP(if-range) + 1

NP (if-else) = NP(expr) + NP(if-range) + NP(else-range)

NP (while) = NP(expr) + NP(while-range) + 1

NP (do-while) = NP(expr) + NP(do-range) + 1

NP (for) = NP(for-range) + NP(expr1) + NP(expr2) + NP (expr3) + 1

NP ("?") = NP (expr1 + NP(expr2) + NP(expr3) + 2

NP (repeat) = NP(repeat-range) + 1

NP (switch) = NP(expr) + $\sum_{i=1}^{i=n} NP(case - range)$ + NP (default-range)

NP (function call) = 1

NP(sequential) = 1

NP(return) = 1

NP(continue) = 1

NP(break) = 1

NP(goto label) = 1

NP(expressions) = Number of && and || operators in Expression

Execution of Path Expressions (complexity expression) are expressed, where "N" represents the number of statements in the body of component (function and "NP (Statement)" represents the acyclic execution path complexity of statement i. Where "(expr)" represents expression which can be derived from flow-graph representation of the statement. For example NPATH measure as follows:

Void func-if-with-assignment (int c)

```
{
  int a=0;
  if(c)
   {
    a=1;
   }
}
```

The Value of NPATH is 2 as follows:

NP(if)=NP(if-range)+NP(expr) + 1

3. McCabb'e Cyclomatic Complexity [10]

Cyclomatic Number is one of the metric based on not program size but more on information/control flow. It is based on specification flow graph representation developed by Thomas J Mc Cabb in 1976. Program graph is used to depict control flow. Nodes are represent processing task (one or more code statement) and edges represent control flow between nodes. McCabe's metrics[10] is example of control flow metrics. To compute cyclomatic complexity V(G) as following methods:

1. For graph G with N vertices(nodes), E edges and P connected components,
   V(G)=E-N+2p
2. V(G)= Total number of bounded area +1
   Where bounded area is in program's CFG, any region encoded by nodes and edges.
3. Number of decision statement of the program +1 or number of predicate node+1

The problem with McCabb's Complexity is that, it fails to distinguish between different conditional statements (control flow structures). Also does not consider nesting level of various control flow structures. NPATH, have advantages over the McCabb's metric [7].

4. Halstead Software Science [8]

Another alternative software complexity measures have to be considered. M. Halstead's Software science measures [8] are very useful. Halseatd's software science is based on a enhancement of measuring program size by counting lines of code. Halstead's metrics measure the number of number of operators and the number of operands and their respective occurrence in the program (code). These operators and operands are to be considered during calculation of Program Length, Vocabulary, Volume, Potential Volume, Estimated Program Length, Difficulty, and Effort and time by using following formulae.

n_1 = number of unique operators,

n_2 = number of unique operands,

N_1 = total number of operators, and

N_2 = total number of operands,

Program Length (N)=N1+N2

Program Vocabulary (n)=n1+n2

Volume of a Program (V)=$N*\log_2 n$

Potential Volume of a Program

(V*)=$(2+n_2)\log_2(2+n_2)$

Program Level (L)=L=V*/V

DOI: 01.IJIT.01.02.158

Program Difficulty (D)=1/L
Estimated Program Length (N)=$n_1 \log_2 n_1 + n_2 \log_2 n_2$
Estimated Program Level (L)=$2n_2/(n_1 N2)$
Estimated Difficulty (D)=$1/L = n_1 N_2 / 2n_2$
Effort (E)=V/L=V*D= $(n_1 \times N_2) / 2n_2$
Time (T)=E/S ["S" is Stroud number (given by John Stroud), The constant "S" represents the speed of a Programmer. The value "S" is 18]
One major weakness of this complexity is that they do not measure control flow complexity and difficult to compute during fast and easy computation.

III. OUR METHODOLOGY

Our method deals with analysis of software complexity metrics for regression testing. We have considered four program characteristics from the literature that are responsible for complexity measures. viz LOC, NPATH, MCC, and HSS. For this study, we have selected only program written in C language given in Figure 1 and Figure 2. The structure of a program P ad P' can be represented by a control flow graph in figure 3, G(P)={N,E,s,e}, where N is a set of nodes representing basic blocks of code or branch points in the function; E is a set of edges representing flow of control in the function; s is the unique entry node and e is the unique exit node

A. Methods:

There are four steps can be used to collect the data as described below. We compute weights of software complexity metrics for original program P and modified program P' (required element):

Step 1: Compute LOC by counting frequency of line numbers.
(i) Blank lines and
(ii) Commenting lines are ignored.

Step 2: Calculate the NPATH complexity measures also known as path count metrics.

Step 3: Compute McCabe complexity.

Step 4: Count the Halstead's Software science of software primitives.

B. Selected Metrics

We have measured LOC, NPATH i.e. acyclic execution paths through components for in an attempt at program optimization, McCabe and finally Halstead's software science complexity metrics. While counting a number of instructions (source), line used for blank and commenting lines are ignored. NPATH measures the acyclic execution paths which counts the number of execution path through a functions. Halstead's metrics measure the number of number of operators and the number of operands and their respective occurrence in the program (code). These operators and operands are to be considered during calculation of Program Length, Vocabulary, Volume, Potential Volume, Estimated Program Length, Difficulty, and Effort and time. For McCabb's complexity measures program graph is used to depict control flow. Nodes are representing processing task (one or more code statement) and edges represent control flow between nodes. Consider an example, Let P be the old version of program and P' be the new version of program in C given below:

Consider a program from figure 1. the complexity measured by us and computed the complexity of the other proposed measures i.e Line of Code (LOC), NPATH Complexity (NC), McCabb's complexity (MCC) and Halstead's software science (HSS). In Figure 2, we have done some modification to the given example of program P i.e some lines are added in the existing program. Some changes are made for regression testing in the existing program and calculated measures from both the programs P and P' in Table 1 and Table II as follows:

```
    #include<stdio.h>
    void main()
1:  {
1:   int a,b,c,n;
1:  scanf("%d %d", & a,&b);
2:  if (a < b)
2:  {
3:  c = a;
3:  }
3:  else
3:  {
4:  c = b;
4:  }
5:  n = c;
6:  while ( n < 8 )
6:  {
7:  if ( b > c )
7:  {
8:  c = 2;
8:  }
8:  else
8:  {
9:  n = n + c +7;
9:  }
10 :  n = n + 1;
10:  }
11:  Printf("%d%d%d",a,b, n);
11.  }
```

Figure 1. Source Program P

```
    #include<stdio.h>
    void main()
1:  {
1:   int a,b,c,n;
1:  scanf("%d %d", & a,&b);
2:  if (a < b)
2:  {
3:  c = a;
3:  }
3:  else
3:  {
4:  c = b;
4:  }
5:  n = c;
6:  while ( n < = 8 )
6:  {
7:  if ( b > c )
7:  {
8:  c = 2;
8:  }
8:  else
8:  {
9:  n = n + c +7;
9:  if ( n % 7 == 0 )
9:  {
10:  c = c + 2;
```


DOI: 01.IJIT.01.02.158

```
10:  }
10:  else
10:  {
11:  c =- -;
11:  }
11:  }
12:  n =++;
12:  }
13:  Printf("%d%d%d",a,b, n);
13:  }
```

Figure 2. Modified Program P'

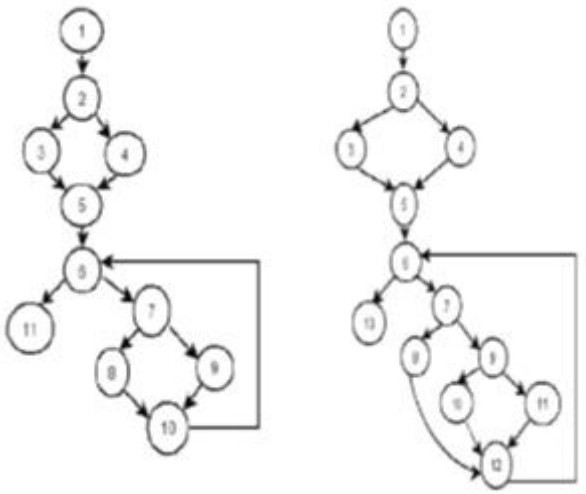

Figure 3.Control Flow Graph for Program P and P'

TABLE I. COMPUTED WEIGHTS SOFTWARE COMPLEXITY MEASURES FROM PROGRAM P

LOC	NPATH	McCabe		Halstead Software Science	
26	6	Nodes	11	n1	13
				n2	8
		Edges	13	N1	46
				N2	29
		Predicate Nodes	3	Vocabu Lary	21
				Program Length	75
		Regions	3	Difficulty	24
				Effort	7567
		V(G)	4	Time	7

TABLE II. COMPUTED WEIGHTS SOFTWARE COMPLEXITY MEASURES FROM PROGRAM P'

LOC	NPATH	McCabe		Halstead Software Science	
34	8	Nodes	13	n1	18
				n2	8
		Edges	16	N1	56
				N2	33
		Predicate Nodes	4	Vocabu Lary	26
				Program Length	89
		Regions	4	Difficulty	37
				Effort	15466
		V(G)	5	Time	14

C. Equivalent Size Measures (ESM):

Equivalent size measures is code redundancy inside a program may be viewed as the intra-program re-use of code. This measure is used by S.D Conte in the year of 1986 [6]. The equivalent size measure is basically effort required to develop software with new code and re-use code is equivalent to the effort of developing the same software. During software development, most of the time re-uses software from the previous code/program. It has been reported that, at IBM's Santa Teresa Laboratory, 77% of all program code is written in place to add new code to existing code [6]. Formula proposed by Bailey and Basili [6] is used to compute the equivalent size measure as follows:

Se=Sn+.02* Su

Where, Se is equivalent size measure

Sn is a measure for newly written code

and Su is a measure for redundant/re-use code (adopted from existing code)

Considering a program P and P', we are able to extract the following equivalent size measures:

Su=24 (i.e. consider as redundant/ re-use code)

Sn= 34-24 (i.e .the total lines of code subtract the lines of redundant/re-use code)

Se= Sn+0.2*Su

= 10+0.2*24

=15 (LOC)

Equivalent size measure (ESM) may be helpful during measurement of software maintenance phase. In this paper, we work with four program characteristics measures. From the above table I and Table II, it is also identified that change behavior of program characteristics

IV. CONCLUSION

Software complexity metrics have a propensity to be used in judging the quality of software development and metrics are relatively easy to generate. The size, complexity and importance of today's software systems demand the application of effective testing techniques. In addition, it was observed that software complexity metrics which enables the tester to counts the acyclic execution paths through a component and improve software productivity and software quality. This approach could be lead to reduce software development cost and improve testing efficacy and software quality. Software metrics with Lines of Code (LOC), Npath (NC), McCabb's complexity metrics (MCC) and Halstead's Software science (HSS) are calculated and observed the change behavior of the code. Finally, the evaluated values from both P and P', the changed behavior of code is identified and tester can use this approach to execute test case and improve software productivity and software quality. In the future study, more complicated program has to be measured with other attributes (complexity measures).

DOI: 01.IJIT.01.02.158

REFERENCES

[1] Shin Yoo & Mark Harman, "Regression Testing Minimization, Selection and Prioritization - A Survey", Technical Report TR-09-09

[2] Thomas J.McCabe "A Complexity Measure" ,IEEE Transactions on Software Engineering, Vol, Se-2,1976.

[3] T.De Marco; Controlling Software Projects; Prrntice Hall, New York,1982.

[4] Israel Herraiz, Jesus M. Gonzalez-Barahona, Gregorio Robles," Towards a theoretical model for software growth" in 29th International Conference on Software Engineering Workshops (ICSEW'07).

[5] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, 2nd Ed.,1990.

[6] S.D. Conte, H.E Dunsmore, and V.Y. Shen. Software Engineering Metrics and Models. Benjamin/Cummings Publishing Company, Inc., 1986.

[7] B.A. Nejmeh. NPATH: A Measure of Execution Path Complexity and Its Applications. Comm. of the ACM, 31(2):188-210, February 1988.

[8] M. Halstead. Elements of Software Science. North-Holland,1977.

[9] K.K Aggarwal & Yogesh Singh "Software Engineering" New Age International, 2003

[10] T.A. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 2(4):308-320, December 1976

[11] A.Fitzsimmons and T. Love;" A Review and Evaluation of Software Science",Computing Survey,Vol.10,No.1 March ,1978

[12] Norman Fenton, " Software Measurement: A Necessary Scientific Basis "IEEE Transaction on Software Engineering, Vol.20,No.3, March,1994,

[13] Software Engineering – A Practitioner's Approach, Roger S. Pressman; McGraw-Hill International Edition.

[14] Rajib Mall, "Fundamentals of Software Engineering", Prentice Hall

DOI: 01.IJIT.01.02.158